# Efficient I/O of Grid Hierarchies for AMR Computations on Parallel Disks

S. Kuo, M. Winslett, Y. Chen, and Y. Cho
Computer Science Department
University of Illinois
{s-kuo,winslett}@cs.uiuc.edu

## Abstract

*Adaptive mesh refinement (AMR) is a promising computational approach that investigates a scientific problem using grids of different resolutions in different portions of the problem space. This allows the more "interesting" areas of the problem space to be investigated in more detail than other areas. In this paper, we present efficient ways for AMR-type applications to do input/output on the IBM SP2. We propose a disk layout that minimizes I/O costs for checkpoint/restart operations and investigate several allocation approaches to decluster grid data to multiple disks to enhance I/O parallelism for timestep/visualization operations. The experiments on the SP2 show that our strategies can help to shorten the I/O time to meet the performance requirements of the two types of I/O operations.*

## 1. Introduction

Scientific applications often center around large multidimensional arrays and are I/O intensive, with the need to efficiently store and retrieve array data. For example, a long-running simulation code periodically outputs snapshots of the state to allow later analysis for trends, and also does checkpoint/restart operations. Managing a large dataset on a sequential machine is not an easy task, and is more complicated in a parallel environment, often exceeding the expertise of scientists. Thus, there is a need for a specialized DBMS or I/O library, which is efficient and easy to use, to facilitate storage management on parallel machines.

Adaptive mesh refinement (AMR) is becoming an important method for solving large scientific problems, like galaxy simulation or crack propagation. Instead of simulating the whole region using a fixed resolution, the AMR technique dynamically allocates computational resources to the more interesting areas of the problem space and allows them to be investigated in a higher resolution than other areas. Therefore, this method facilitates tracking of local phenomena, like the shock waves in computational fluid dynamics [2]. The underlying data structure of the AMR technique is a hierarchy of grids which changes dynamically, instead of a single multidimensional array. In terms of I/O, the new computational model and data structure require new organizations for data on disk to allow fast storage and retrieval.

In this paper, we focus on optimizing physical schemas in order to minimize the I/O time for the two most common types of I/O operations issued by AMR-type applications - checkpoint/restart and timestep/visualization. For long-running production runs, it is desirable to save the state of certain arrays periodically (checkpoint) in order to resume (restart) from a previous state in case of a system failure. Because checkpoint operations are performed frequently, a good physical schema is required to minimize the I/O time. We previously proposed the use of "natural chunking" [18] for checkpoint schemas in applications using High Performance Fortran-style data distributions, which spread data evenly across all processors. For AMR-type applications, a naive natural chunking schema cannot guarantee high performance and an improved data layout is required to speed up the I/O process.

Another common set of I/O operations is associated with timestep/visualization computations. For time-dependent applications, snapshots of certain arrays are output at selected intervals over time. Output data will then be analyzed by visualization tools. Again, the performance metric for these operations is the response time - minimizing the time to store the whole dataset in a timestep operation and the time to retrieve a subspace from the dataset, like a multidimensional range query, in a visualization operation. In a parallel environment, an effective way of minimizing the response time for data retrieval is to enhance I/O parallelism - decluster the dataset among the disks in a way such that every future retrieval can be balanced among the disks. Declustering has been a hot research topic in spatial databases [6, 7, 12, 13, 15]. However, the previous work

This research was supported by NASA under NAGW 4244 and NCC5 106, and by DOE under B341494. Computing facilities were provided by Argonne National Laboratory.

assumed that the declustering algorithm and data loading were run off line. Our goal is to optimize both timestep and visualization operations on line and we are seeking a declustering algorithm which minimizes the computation time as well as produces high-quality solutions.

In the remainder of the paper, we begin by giving an overview of the AMR method and the Panda array I/O library, which is used as a testbed for verifying various physical schemas. Section 3 proposes a new physical schema designed for checkpoint/restart operations. Section 4 describes various declustering algorithms for efficient timestep/visualization. Preliminary results of the experiments conducted on an IBM SP2 are shown in Sections 3 and 4. Related work is discussed in Section 5 and Section 6 concludes this paper.

## 2. Background

### 2.1. The adaptive mesh refinement method

AMR is a computational approach which provides an effective way of deriving accurate solutions to application problems while using fewer computational resources, like CPU cycles and memory. AMR has received much attention from computational scientists in recent years and several libraries [1, 10, 14, 20] have been developed to ease the task of application scientists who wish to use this method.

The concept is as follows: AMR starts with a coarse grid (level 0) with a minimum acceptable resolution which covers the entire computational domain. As the computation progresses, grid points that the application identifies as "interesting" are tagged and grouped into a rectangular region. A finer grid (level 1, a child of the level 0 grid) with a higher resolution corresponding to the tagged region is then overlaid on the original coarse grid. The refinement process proceeds recursively until all solution points are represented with an acceptable resolution.

The data structure corresponding to the AMR method is a hierarchy of nested grids which changes dynamically at run-time. Figure 1 shows an example hierarchy. When running on parallel machines, the whole grid hierarchy is distributed to multiple processors and each processor is responsible for the computation of a set of component grids, where each component might be a single grid or a subspace of a grid in the grid hierarchy, depending on the decomposition strategy used. In this example, all grid points having the same coordinates but in different levels are assigned to the same processor in order to maintain parent-child locality. However, many other distribution methods [10] do not have this property.

Partitioning the grid hierarchy across multiple processors is not an easy task. Several constraints have to be met simultaneously in order to minimize the total execution time.

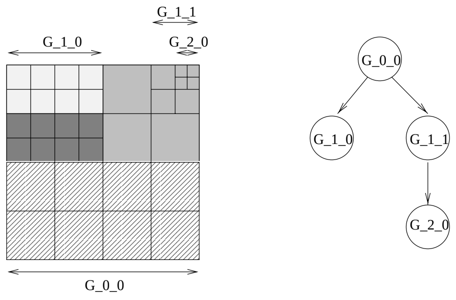


**Figure 1. Example of a grid hierarchy with four grids belonging to three different levels. A grid abbreviated as $G\_m\_n$ is read as "grid $n$ in level $m$". This example uses a space-filling curve enumeration to distribute grids to four processors [14]. Rectangles filled with different patterns are assigned to different processors.**

First, the computation load should be balanced. A special characteristic of an AMR algorithm is that achieving memory balance, i.e. assigning equal amounts of data (grid points) to each processor, does not necessarily guarantee CPU-time balance. This is because in an AMR algorithm, the computational load of a grid is determined both by the number of grid points in the grid and the level of the grid in the AMR hierarchy. Finer resolution grids are updated more frequently than coarser ones. Therefore, if the load is perfectly memory balanced, it may happen at run-time that one processor has many more fine-grids than another. This leads to a huge load imbalance where all of the other processors must wait on the overloaded one to finish. On the other hand, if the load is well CPU-time balanced, one processor may have many coarse-level grids which require much more memory than processors with an equivalent load of fine-level grids.

Second, the communication overhead should be minimized. There is a special type of communication, called inter-grid communication, for AMR algorithms in addition to normal message passing of boundary information. Inter-grid communications are used to propagate solution values along the hierarchy. Parent processors communicate with child processors to propagate solution values. Therefore, the distribution for the grid hierarchy is aimed at balancing the load as well as maintaining the parent-child locality.

In general, the outcome of the various partitioning strategies [10, 14] to distribute the load to multiple processors for AMR-type applications is that grid data are normally unevenly distributed among compute processors in order to simultaneously meet the constraints described above.

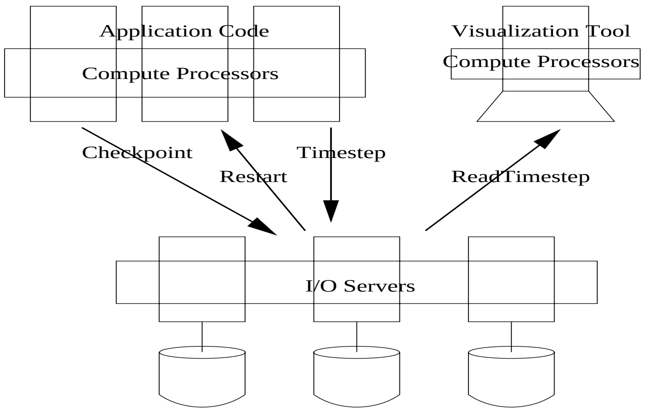


**Figure 2. System architecture of Panda**

## 2.2. The Panda array I/O library

Panda (http://drl.cs.uiuc.edu/panda/) is a DBMS-style I/O library developed at the University of Illinois to facilitate storage management for scientists. It is designed to support SPMD-style application programs running on distributed and shared memory architectures. Figure 2 shows an environment where Panda is running. As shown in the figure, scientific applications and visualization tools are running on Panda clients, which communicate with Panda servers when they want to input or output array data.

Panda provides users with a set of high-level I/O interfaces - `checkpoint`, `restart`, `timestep`, and `read-timestep`. With these APIs, applications can read and write entire arrays as well as arbitrary rectangular regions of arrays directly. For each I/O operation, users describe the arrays to be read/written - ranks, sizes, memory locations, the distributions of the arrays among Panda clients and the desired arrangement of array data on Panda servers[1]. Both distributions can be described in an HPF (High Performance Fortran) format [17], or in a more general format (for AMR-type applications). When the distribution of data among processors is irregular, an intuitive way of describing the distribution is to use a general block format where each grid component is represented by its origin and size. That is, an $n$ dimensional block is characterized by two vectors, $base = [b_0, \ldots, b_{n-1}]$ and $size = [s_0, \ldots, s_{n-1}]$. A linked list of blocks is used to represent the grid components belonging to a compute processor.

At roughly the same time, the application processes running on the compute processors issue a request to each local Panda client to input or output a set of arrays. Each client sends to a selected server (master server) a high-level description of the request (*schema message*) containing its block information. Once a server receives the schema messages, it informs all other servers of the schema information. Based on the schema information, if the operation is a write, each server can compute an optimal disk organization and determine which portions of the array data are its responsibility. Then it plans how to request array data from clients and writes gathered data to disk, with the goal to optimize disk accesses. We call this architecture "server-directed I/O", described in detail in [17].

[1]The specification of a disk layout is optional. If users do not specify the desired organization of data on disk, Panda automatically chooses a layout suitable for the I/O operation.

For AMR-type applications, as the distribution of grid data is irregular, it is harder for users to decide the optimal organization of data on disk, compared to HPF-style applications. In this paper, we assume that users do not specify the desired schema and leave it to Panda with the goal to minimize the I/O time.

## 3. Checkpoint-Restart Operations

### 3.1. Approaches

In a distributed-memory context, a checkpoint operation requires the array data distributed across multiple processors to be stored in files. Seamons [18] proposed a physical schema, called natural chunking, which was shown to provide high performance for distributed checkpoint operations. Based on the assumption that most checkpoint files will never be read again, it uses the data's in-memory distribution as the on-disk distribution. That is, I/O server $n \; mod \; m$, where $m$ is the number of I/O servers, is responsible for outputting the data belonging to compute processor $n$. Figure 3-(b) shows an example natural chunking layout.

For HPF-style applications, balancing the CPU-time load usually conforms to balancing the memory load. Therefore, the amount of data assigned to each compute processor is normally balanced in order to minimize the total computation time. As the data is balanced among the processors, the natural chunking strategy guarantees a balanced load among the I/O servers. However, for AMR-type applications, as discussed in Section 2.1, balancing the CPU-time load and the memory load do not necessarily conform to each other. Also, there is an extra constraint to maintain the communication locality. Therefore, the natural chunking strategy proposed in [18] (called "naive natural chunking" in this paper) cannot guarantee a balanced load for I/O servers. A new physical schema is required.

We propose an improved organization, called "partitioned natural chunking", for AMR-type applications to do checkpoints/restarts. It follows the idea of the naive natural chunking to use the same in-memory distribution as on-disk distribution in order to minimize any overhead of data reorganization, while at the same time balancing the load among the I/O servers. The algorithm first enumerates all grids belonging to all compute processors across all levels and then decomposes the enumeration into $m$ partitions with equal amounts of data. Each partition is then assigned to an I/O server. The enumeration is ordered by processor id (from the smallest to the largest). For some data distributions, altering the enumeration order will result in a slightly better performance during I/O by having the cuts exactly at the grid boundaries. However, the overhead of finding the best enumeration order might be larger than the performance gain. Figure 3-(c) shows an example.

By using the partitioned natural chunking strategy, the amount of data to be written by each I/O server is balanced. Also, reconstructing the compute image in a restart operation with the same number of processors is easy and efficient: first, only at most $m - 1$ grids have been split across I/O servers in a checkpoint operation and need to be recomposed. Second, recomposing a grid spread across multiple I/O servers is an easy task as only concatenation is required.

We also experimented with another physical schema for comparison. In the comparison layout, each grid is decomposed into $m$ parts, and each portion is assigned to an I/O server, as shown in Figure 3-(d). This approach can balance the load. As each grid is decomposed across all I/O servers, if each server gathers/scatters the grids using the same order, there will be contention between the servers for transferring each piece from the compute processors, resulting in high communication costs. We reduce the contention problem by using a different gathering/scattering sequence for each server - I/O server $n$ starts with the grids belonging to compute processor $n$. However, the shift strategy does not guarantee contention free data transfer and we expect this layout to have lower message passing throughput.

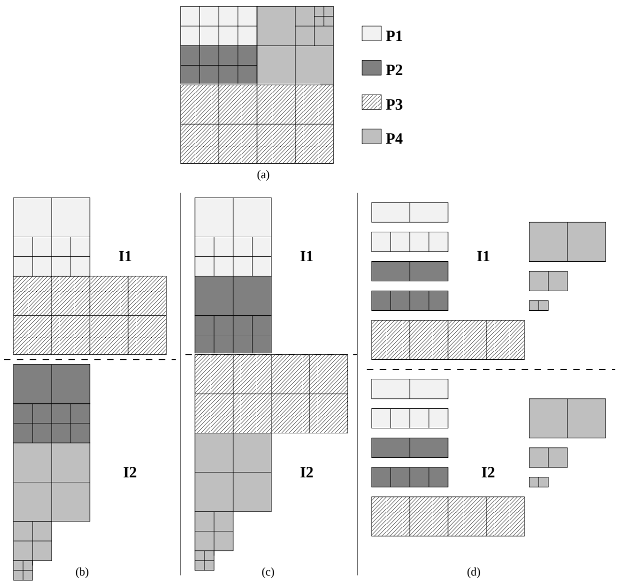


**Figure 3. Different organizations of a grid hierarchy on disks. (a) The grid hierarchy shown in Figure 1, with four compute processors and two I/O servers. Rectangles filled with different patterns are assigned to different processors. (b) The “naive natural chunking” strategy. Compute processors are assigned to I/O servers in a round-robin order. (c) The “partitioned natural chunking” strategy. Grids are first organized into a linear structure and then decomposed into two partitions of equal size to be assigned to two I/O servers. (d) An experimental organization used for comparison. Each grid is decomposed into two portions to be assigned to two I/O servers.**

## 3.2. Experimental validation

### 3.2.1. Experimental setups

The experiments described in this paper were conducted on an SP2 at the Argonne National Laboratory (ANL). Table 1 is a brief summary of the current configuration of the ANL SP2. The JFS performance numbers were determined empirically following the methodology of [17]. More details on the SP2 installation can be found on line at http://www.mcs.anl.gov/CCST/computing/quad/.

We wrote a generator to produce AMR-type distributions, which are used as input to Panda. The input to the generator is a collection of user-selected grids of different resolutions which represent the grid hierarchy generated by AMR-type applications[2]. The generator uses a decomposition strategy based on space-filling curve [14] to partition and distribute the problem space to compute processors.

### 3.2.2. Performance results

Figure 4 shows an example distribution generated by the simulator. As can be seen from the figure, the CPU load[3] is nearly balanced among the compute processors; however, the amount of data is not. This is typical of an AMR-type application as explained in Section 2.1.

[2]At first, we used the DAGH infrastructure [14] and an available real AMR-type application, an implementation of the Buckley-Leverette equations in 2D, to generate the grid hierarchy. However, the dataset generated by the application is too small (less than 1 MB) to show any performance difference of using different physical schemas. Also, the application got stuck (the computation time was longer than 3 hours and we later killed the job) when the problem size was slightly enlarged. We are still looking for a more suitable application.

[3]The CPU load is computed by aggregating the computational load of all grid points belonging to a process. The computational load of a grid point is determined by the frequency that it is updated, assuming that each higher level child grid updates twice as often as its parent grid.

| | |
|---|---|
| **General information** | |
| Processor | 120 MHz POWER2 |
| Main memory | 256 MB |
| Scratch space | 2 GB |
| **Measured file system peak performance** | |
| JFS writes | 6.1 MB/sec |
| JFS reads | 7.1 MB/sec |
| **Message passing performance** | |
| Latency | 31 microseconds |
| Bandwidth | 90 MB/sec |

**Table 1. The current hardware and software configurations of the ANL SP2**

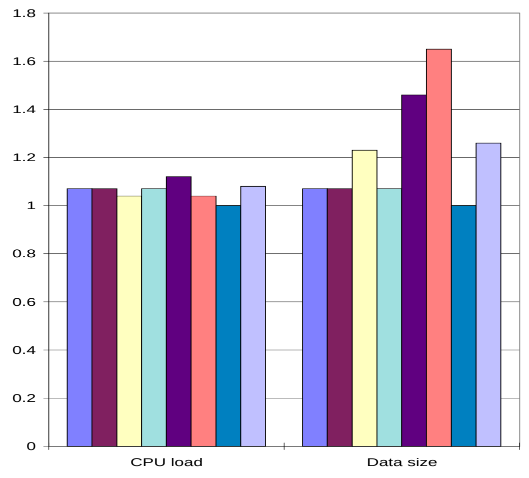


**Figure 4. Example output of the simulator. The left-hand side group of bars shows the CPU load of each processor (normalized to the smallest value). The right-hand side group shows the amount of data residing in each processor's memory, again normalized to the smallest value. The total size of the grid hierarchy is 512 MB. This distribution is used in all experiments shown in this paper.**

Figure 5 shows the performance of checkpointing and restarting the grid hierarchy shown in Figure 4. All data points are the average of 4 runs, and the error bars correspond to a 95 percent confidence interval for the mean. As expected, the naive natural chunking strategy cannot guarantee high performance as the amount of data input/output by each I/O server varies greatly; Figure 6 shows this trend clearly. The partitioned natural chunking strategy boosts the performance to be over 95% of the peak throughput of the underlying file system at each I/O server for all configurations.

When the comparison approach is used, though data is evenly assigned to the I/O servers, performance drops a little bit compared to the partitioned approach when 2 or 4 I/O servers are used and degrades up to 5% when the number of I/O servers is increased to 8 for checkpoint operations. Communication contention is responsible for the slowdown. Figure 6 verifies this explanation by showing the costs of communications during writes with simulated infinitely fast disks (created by commenting out all `fread` and `fwrite` calls). Panda's message passing throughput drops up to 30% due to the communication contention. For the overall I/O time, the effect of communication overhead is small when a small number of I/O servers is used. But when 8 I/O servers are used, the effect is visible. In a workstation cluster, where the network may be relatively slow, minimizing communication overhead is particularly important. As Figure 6 illustrates, we expect the partitioned natural chunking approach to be more robust across platforms than the comparison approach as it takes both the message passing performance and workload balance into consideration.

## 4. Timestep-Visualization Operations

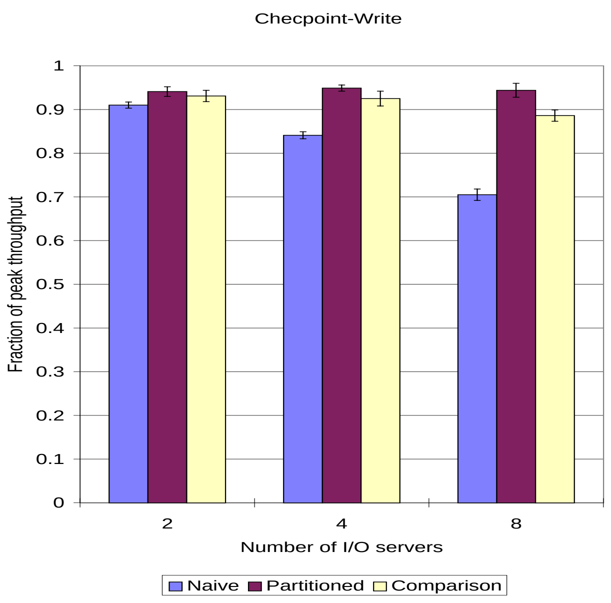
### 4.1. Approaches

Applications solving time-dependent problems output data at selected intervals over time (timestep data), which will be later post-processed for eventual visualization by visualization tools. The physical schema for timestep data on multiple disks should be optimized for both timestep-write (abbreviated as T-W in this paper) and visualization-read (abbreviated as V-R) operations, for which I/O time can be a significant fraction of total run time.

The read operations performed during visualization[4] usually correspond to multidimensional range queries and an important performance factor is the degree of I/O parallelism when running in a parallel environment. Therefore, an optimal layout for timestep output data is to decompose the array into chunks (tiles) and then distribute chunks to I/O servers with the goal that every future retrieval can be balanced among the disks and use maximum parallelism. Applying the same chunking strategy to a grid hierarchy, one can decompose each grid in the grid hierarchy and assign the chunks belonging to each grid independently with the assumption that if all grids are distributed in an optimal way, every future retrieval of a subspace of a grid hierarchy is likely to be balanced among the disks.

The assignment of chunks conforms to a declustering

[4]In a grid hierarchy, as fine grids are overlaid on coarser ones, many grid points will have the same coordinates. Therefore, when viewing a subspace of a grid hierarchy, users can specify to view grid points of a specific level. If users do not specify it in the input, for each point, the grid point of the finest-grained level will be viewed.



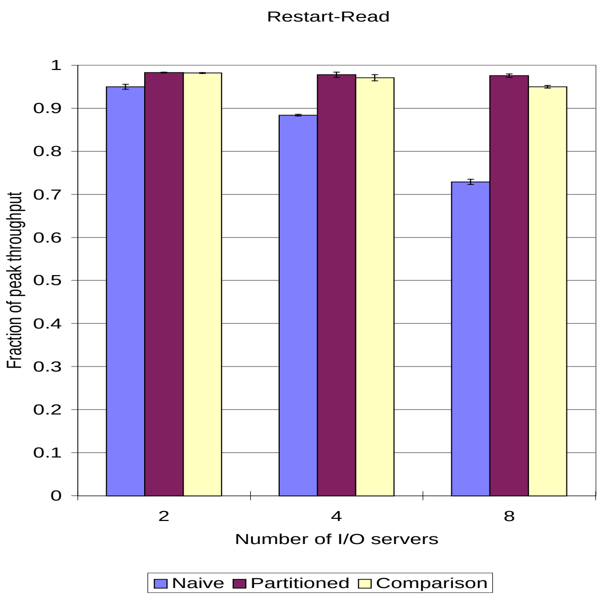


**Figure 5. Throughputs for reading/writing the 512 MB grid hierarchy of Figure 4 with 8 compute processors and 2, 4 or 8 I/O servers. The upper graph shows the average fraction of peak file system throughput utilized by each I/O server during checkpoint operations and the lower graph shows the fraction of peak throughput for restart operations.**

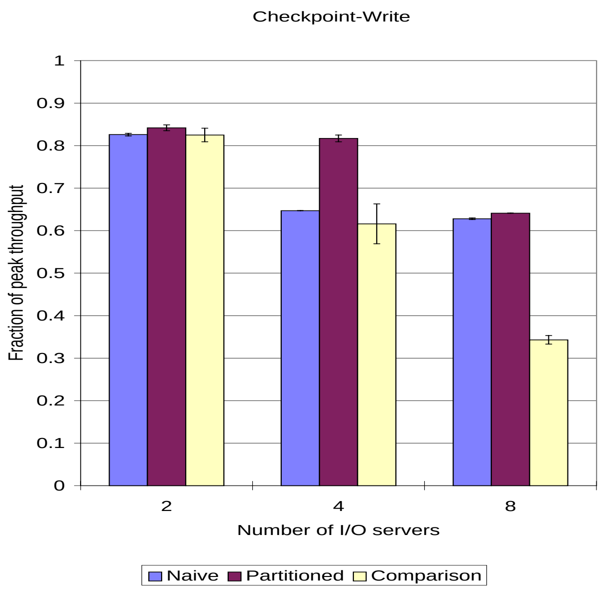


**Figure 6. Comparison of three layouts with simulated fast disks, using the same configuration as in Figure 5. The graph shows the average fraction of peak message passing (MPI) throughput utilized by each I/O server for checkpoint operations. Results for restart operations are similar.**

problem - given a set of data blocks (chunks) and $m$ disks, define a mapping between data blocks and disks [12]. In recent years, various declustering algorithms have been proposed for spatial databases. In general, these methods can be categorized into three types: linear-time, similarity based and graph-coloring. Du [6] proposed the *disk modulo* method to assign data blocks to disks in a round-robin manner. The advantage of this approach is its simplicity. Faloutsos [7] applied the idea of the space filling curve to impose a linear ordering on multidimensional data blocks. The data blocks are then assigned to disks in a round robin fashion. This approach has been empirically shown to perform better than the disk modulo method for square queries. All the above are linear-time approaches as the assignment can be computed in approximately linear time; however, the assignments these approaches generate are generally suboptimal. Also, these approaches are less flexible in extending to handle heterogeneous disks.

The similarity based declustering approaches map the declustering problem into a min-cut graph partitioning problem. A fully-connected, complete graph is first created by mapping each data block to a node in the graph. The weight of an edge connecting two nodes is determined by the probability of two corresponding blocks being accessed together in data retrieval. Thus, the problem of finding the optimal allocation corresponds to finding the minimal-cut partition of the graph. Liu [12] applied the Kernighan-Lin algorithm (a specialized simulated-annealing algorithm for solving the graph partitioning problem) to find the minimal-cut partition. The solution it generates is generally good but it takes long to converge. Another similarity approach, called the minimax spanning tree algorithm, was proposed in [13]. Its key idea is to extend Prim's minimal spanning tree algorithm to generate partitions. The execution time of the minmax approach is shorter than applying the Kernighan-Lin search algorithm but the solution is less optimal. In general, the drawback of similarity based declustering algorithms are their high complexity - at least quadratic for the mapping and the optimization phases, resulting in long execution time. However, these approaches are quite flexible in taking other constraints (besides minimal-cut) into concern during partitioning.

Recently, Prabhakar mapped the declustering problem to a graph coloring problem [15]. Again, a complete graph is first created as in the similarity based approach. Then, instead of finding the minimal cut by taking global assignments into concern, it only optimizes local assignments by preventing a data block to be assigned to the same disks as its close neighbors. Therefore, declustering data blocks to $m$ disks corresponds to a $m$-colorability problem: for each node in the graph, find its closest $m$-1 neighbors and call them adjacent nodes; use $m$ colors to color the graph such that no two adjacent nodes are assigned with the same color. This approach cannot generate optimal solutions but is more computationally efficient than similarity based algorithms.

All the above algorithms can be used in Panda to assign chunks to disks. Linear-time approaches require less computation time and can give good performance for T-W operations; however, the solutions they generate are suboptimal for range queries and the approaches are less flexible for extensions. Similarity based approaches produce high quality solutions but require more computation time during a T-W operation, which will be visible in application response time. The graph-coloring approach sits in the middle. The solutions it generates are not as good as the similarity based approaches but it requires less computation time and is as flexible as similarity based approaches. Flexibility is important for our future extensions to handle heterogeneous disks and to consider the correlations of assigning multiple grids, e.g. chunks having the same coordinates but in different levels can be assigned to the same disks as the chance that they will be visualized together is very low. Due to these concerns, we devised a solution based on graph coloring (which we later discovered to be a generalization of [15]).

The pseudo code for our algorithm appears below. The assignment algorithm removes the chunk at the head of the queue and determines which disks are suitable (not the same as its neighboring chunks) for the chunk. When more than one disk is suitable, in [15], the algorithm chooses randomly among those with equal probability. We, in addition, take load balancing into consideration so that among those suitable chunks, the algorithm chooses the least loaded disk (assigned the least amount of data currently). When there is a tie, i.e. several suitable disks are equally loaded, function `choose_disk` resolves the tie. In the current implementation, as close neighbors can be computed easily, more distant neighbors will be checked until the tie is resolved. Other approaches like random assignment can be used if it is time-consuming to find neighbors.

```
Coloring(int num_of_disks) {
 weights[ ] = [0, ..., 0];
 add(queue, first_chunk);
 while (not empty(queue)) {
  current_chk_id = remove_head(queue);
  // choose disk to hold the chunk
  min_weight = MAX_WEIGHT;
  min_weight_disks = { };
  for (i = 0; i < num_of_disks; i++) {
   if (!suitable(current_chunk_id, i))
     continue;
   if (weights[i] == min_weight)
    add_to_set(min_weight_disks, i);
   if (weights[i] < min_weight) {
     min_weight = weights[i];
     min_weight_disks = {i};
   }
  }
  assign(current_chunk_id,
         choose_disk(min_weight_disks));
  // push neighboring chunks to the queue
  for (each neighboring chunk c)
    add(queue, c);
 }
}
```

## 4.2. Performance results

We tested the declustering approaches for T-W/V-R operations on the ANL SP2. In this paper, we only focus on evaluating the performance of different declustering algorithms and use a fixed decomposition strategy and chunk sizes. Each grid in the grid hierarchy is decomposed separately into equally sized partitions along each dimension, creating a set of chunks. Then, all chunks belonging to a grid are assigned in one call to a declustering algorithm. In each declustering algorithm, we allocated a vector called `weights` to record the amount of data assigned to each disk currently. The declustering algorithm takes the vector as an input in assigning chunks of each grid in order to maintain load-balancing across all grids.

We compared four declustering approaches - round-robin, which is ordinarily used in Panda to assign chunks to I/O servers; Kernighan-Lin [12], a similarity based approach that uses simulated annealing to search for an optimal assignment; Hilbert-curve [7], a linear strategy that uses a space-filling curve to enumerate the chunks, and assigns the $n$th chunk in the enumeration to server $n \ mod \ m$[5] and our modified graph-coloring approach. Figure 7 shows the I/O time for both T-W and V-R operations. Each result is an average of 4 iterations and for the V-R operations, the response time is the average of retrieving 16 randomly chosen subspaces. The area of each requested subspace is 1/8 of the original problem size. However, the total amount of data retrieved from disk varies depending on the location of the subspace.

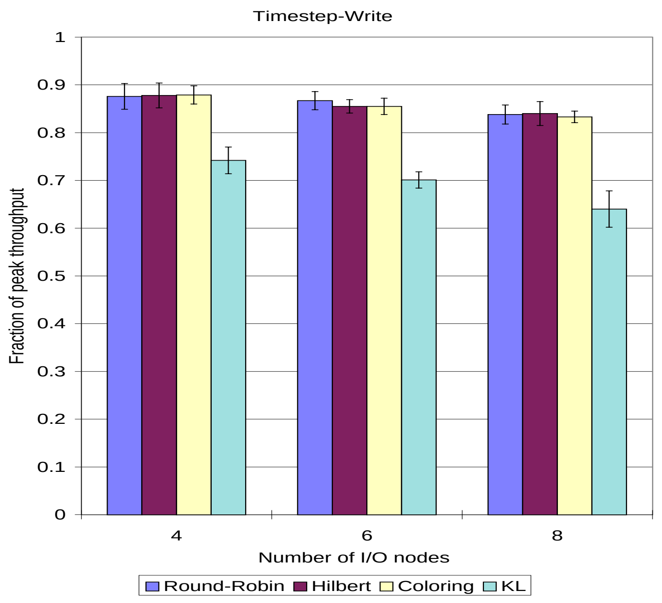
As can be seen from the figures, for T-W operations, all approaches but Kernighan-Lin deliver high performance, utilizing above 85% of the peak file system throughput, as the computation time of these declustering algorithms is negligible. The Kernighan-Lin approach, however, requires high computation cost. For the experiments shown in Figure 7, the computation time is tens of seconds and the throughput drops to as low as 65% of peak.

For V-R operations, the Kernighan-Lin approach produces the best results. The graph-coloring approach is a close second. The next is the Hilbert-curve approach which is a bit worse than the graph-coloring approach as this approach does not intentionally assign close-by chunks to different servers, instead relying on the property of the Hilbert curve to achieve declustering. The round-robin approach performs worse than the others as it does not take the relative locations of chunks into concern to scatter close-by chunks to different servers during the assignment.

Ideally, the implementation of T-W operations should automatically choose the size for chunks on disk as well as the assignment of chunks to disks because both factors are tightly interrelated and have a strong impact on I/O performance. To see the effect of using different chunk sizes, Figure 8 shows the performance of T-W/V-R operations using the graph-coloring declustering approach and varying chunk sizes in each run. For T-W operations, performance degrades by up to 8% as the chunk size decreases, because more chunks mean more time to assign and gather.

For V-R operations, the trends are more complex, as multiple performance factors are involved. In Panda, a chunk is a unit for reads. That is, a whole chunk will be read into memory when part of it is requested. Hence, a smaller chunk size can reduce the overhead of reading unnecessary data and can help to balance the load among the I/O servers more evenly. However, it will cause a larger num-

[5]We implemented both the Kernighan-Lin and Hilbert-curve approaches according to the algorithms described in papers [9] and [7] respectively.



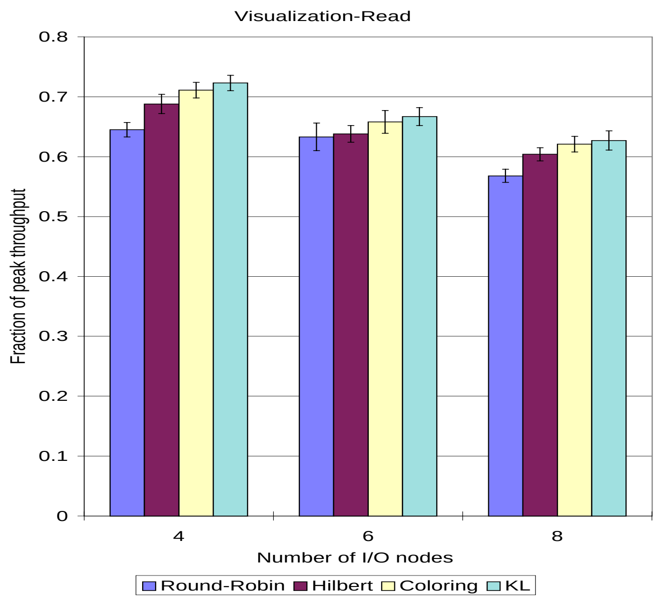


**Figure 7. Performance of T-W operations and V-R operations on a 512 MB grid hierarchy with 8 compute processors and 4, 6 or 8 I/O servers, using different declustering strategies and a fixed chunk size of 512 KB. The upper graph shows the average fraction of peak file system throughput utilized by each I/O server during timestep operations. The lower graph shows the average fraction of peak file system throughput for visualization operations. Read throughput is calculated by dividing the response time by the amount of data requested (which may be smaller than the amount read).**

ber of chunks to be retrieved during a V-R operation and more random seeks will be performed to fetch data chunks, resulting in a longer response time for V-R operations. We plan to investigate the problem of automatically choosing the optimal chunk size for declustering in the future.

## 5. Related Work

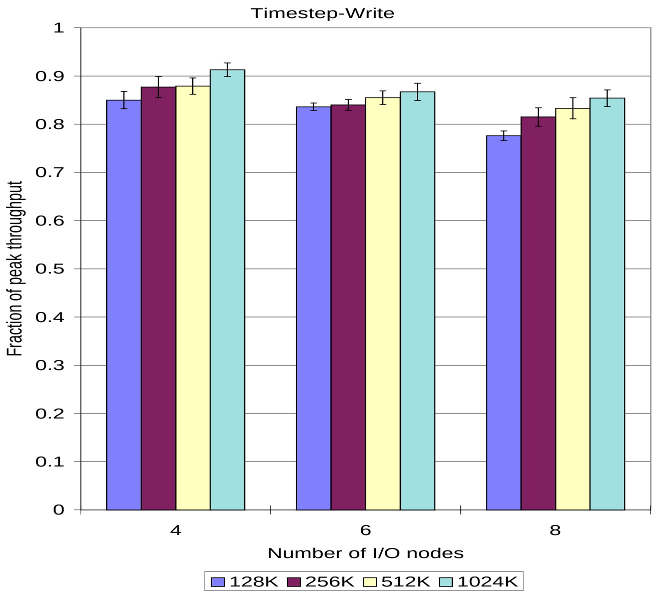
In recent years, several database systems have been designed to handle scientific datasets [3, 5, 16, 18]. Data volume has been the most important characteristic of these databases, forcing the data to be stored on secondary or even tertiary storage devices instead of keeping them in fast memory. Therefore, most of the research has emphasized providing an efficient organization of data on disk in order to speed up users' accesses to data. They argue that the traditional method of storing an array in row-major or column-major order will lead to disastrous performance when access patterns are different from storage patterns.

[8] used a PLOP file structure for the array storage of radio astronomy applications at the NRAO. In PLOP files, a specific set of dimensions is partitioned by splitting each dimension into a series of slices. Then the intersection of one slice from each dimension defines one logical data bucket. This method optimizes query performance when the choices of split are based on a preliminary statistical survey of data access patterns. [16] enhanced the POSTGRES DBMS to support multidimensional arrays with chunked schemas (one chunk per disk block). They chose an optimal chunk layout based on the actual access patterns of the arrays when used by global change scientists in the Sequoia project [19]. Paradise [5] used a client-server architecture and provided an extended-relational data model for modeling GIS applications, with support for 2D chunked arrays with a structure called tiles. [4] showed how to use a combination of simulated annealing and a rule-based approach for on-line generation of optimal I/O plans and layouts using the Panda parallel I/O library for scientific applications. All these layouts worked well with the simple array data, but are not suitable for the grid hierarchy datatype generated and visualized by AMR-type applications.

Declustering is an effective way of achieving disk parallelism, hence reducing the response time of a data retrieval in a visualization operation. Declustering algorithms for array data are described in Section 4.

## 6. Conclusions and future work

This paper has presented efficient physical schemas for fast storage and retrieval of the grid hierarchies generated by AMR-type applications. We proposed a disk layout that minimizes I/O costs for checkpoint/restart operations and



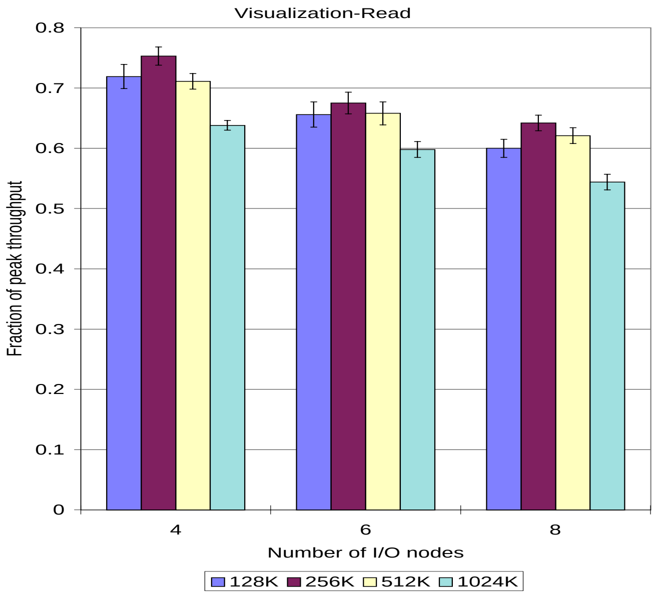


**Figure 8. Performance of T-W operations and V-R operations on a 512 MB grid hierarchy with 8 compute processors and 4, 6 or 8 I/O servers, using different chunk sizes and a graph-coloring declustering strategy. The upper graph shows the average fraction of peak file system throughput utilized by each I/O server during timestep operations. The lower graph shows the average fraction of peak file system throughput for visualization operations.**

showed that our partitioned natural chunking strategy can deliver over 95% of the peak file system throughput on an SP2 in checkpointing/restarting grid hierarchies. We also studied allocation approaches to decluster grid data across multiple disks to enhance I/O parallelism for reads and writes of time-dependent calculations and visualization operations. Experiments on an SP2 showed a graph-coloring declustering strategy gives good quality data declustering and is fast enough to be used on line.

For optimizing the physical schema for timestep and visualization operations, we used a mesh layout to decompose grids into chunks of fixed sizes. As briefly described in Section 4.2, there are various factors in deciding an optimal chunk size. For example, a smaller chunk size can help to balance the load among the I/O servers more evenly but results in longer computation time for the declustering algorithm. We believe that there should be a systematic way to choose an optimal chunk size and shape by taking all these factors into concern and an analytical model might be helpful to determine the optimal chunk size. We plan to work on this problem in the future.

In the past, we have focused on handling heterogeneous parallel disks in Panda [11] and balanced the data among the I/O servers dynamically based on the capability of the servers. AMR-type distributions can be viewed as another kind of heterogeneity for a parallel DBMS - input (data) heterogeneity. The interesting area of the problem changes dynamically and is solved with a higher resolution to allow tracking of local phenomena. We plan to merge these two works together in the future, i.e. store data generated by AMR-type applications in cases where I/O servers have different capabilities.

## References

[1] D. Balsara, M. Lemke, and D. Quinlan. AMR++, a C++ object oriented class library for parallel adaptive refinement fluid dynamics applications. *American Society of Mechanical Engineers, Winter Annual Meeting*, 157, 1992.

[2] M. Berger and P. Colella. Local adaptive mesh refinement of shock hydrodynamics. *Journal of Computational Physics*, 82, 1989.

[3] C. Chang, B. Moon, A. Acharya, C. Shock, A. Sussman, and J. Saltz. Titan a high performance remote-sensing database. In *Proceedings of the 13th International Conference on Data Engineering*, 1997.

[4] Y. Chen, M. Winslett, Y. Cho, and S. Kuo. Automatic parallel i/o performance optimization in Panda. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures, and Distributed Systems*, 1998.

[5] D. DeWitt, N. Kabra, J. Luo, J. Patel, and J.-B. Yu. Client-server Paradise. In *Proceedings of the 20th International Conference on Very Large Data Bases*, 1994.

[6] H. Du and J. Sobolewski. Disk allocation for cartesian product files on multiple disk systems. *ACM Transactions on Database Systems*, 7(11), 1982.

[7] C. Faloutsos and P. Bhagwat. Declustering using fractals. In *Proceedings of the Second International Conference on Parallel and Distributed Computing*, 1992.

[8] J. Karpovich, J. French, and A. Grimshaw. High performance access to radio astronomy data: A case study. In *Proceedings of International Conference on Scientific and Statistical Database Management*, 1994.

[9] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2), 1970.

[10] S. Kohn and S. Baden. A parallel software infrastructure for structured adaptive mesh methods. In *Proceedings of Supercomputing '95*, 1995.

[11] S. Kuo, M. Winslett, Y. Chen, Y. Cho, M. Subramaniam, and K. Seamons. Parallel input/output with heterogeneous disks. In *Proceedings of International Conference on Scientific and Statistical Database Management*, 1997.

[12] D.-R. Liu and S. Shekhar. A similarity graph-based approach to declustering problems and its application towards parallelizing grid files. In *Proceedings of the 10th International Conference on Data Engineering*, 1994.

[13] B. Moon, A. Acharya, and J. Saltz. Study of scalable declustering algorithms for parallel grid files. In *Proceedings of the 10th International Parallel Processing Symposium*, 1996.

[14] M. Parashar and J. C. Browne. Distributed dynamic data-structures for parallel adaptive mesh refinement. In *Proceedings of International Conference for High Performance Computing*, 1995.

[15] S. Prabhakar, D. Agrawal, A. Abbadi, A. Singh, and T. Smith. Browsing and placement of multiresolution images on parallel disks. In *Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems*, 1997.

[16] S. Sarawagi and M. Stonebraker. Efficient organization of large multidimensional arrays. In *Proceedings of the 10th International Conference on Data Engineering*, 1994.

[17] K. Seamons, Y. Chen, M. Winslett, Y. Cho, S. Kuo, and M. Subramaniam. Persistent array access using server-directed I/O. In *Proceedings of International Conference on Scientific and Statistical Database Management*, 1996.

[18] K. Seamons and M. Winslett. Physical schemas for large multidimensional arrays in scientific computing applications. In *Proceedings of International Conference on Scientific and Statistical Database Management*, 1994.

[19] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith. The Sequoia 2000 storage benchmark. In *ACM SIGMOD International Conference on Management of Data*, 1993.

[20] Q. Stout, D. D. Zeeuw, T. Gombosi, and C. Groth. Adaptive blocks: A high performance data structure. In *Proceedings of Supercomputing '97*, 1997.